Hardware Synthesis for Asynchronous Communications Mechanisms

Kyller Gorgônio
Federal University of Campina Grande
Embedded Systems and Pervasive Computing Lab.
Campina Grande, Brazil
kyller@dee.ufcg.edu.br

Jordi Cortadella
Universitat Politcnica de Catalunya
Department of Software
Barcelona, Spain
jordicf@lsi.upc.edu

Abstract

Asynchronous data communication mechanisms (ACMs) have been extensively studied as data connectors between independently timed concurrent processes. In this work an automatic method for synthesis of re-reading ACMs is introduced. This method is is oriented to the generation of hardware artifacts. The behavior of re-reading ACMs is formally defined and the correctness properties are discussed. Then it is shown how to generate the ACMs specifications and how they can be translated into a proper hardware implementation. Verilog has been used as the target language to describe the hardware being synthesized.

1. Introduction

After satisfying design requirements on data, maximizing asynchrony is one of the most important issues when designing communication schemes between asynchronous processes. This task becomes more difficult when the traffic between the processing elements increases. An *Asynchronous Communication Mechanism* (ACM) is a scheme which manages the data transfer between two processes not necessarily synchronized for this purpose. The general scheme of an ACM is shown in Figure 1. A shared memory is used to transfer data and a set of variables is used to control the access to the memory. The data being transferred consists of a stream of items of the same type. The writer and reader processes are single-threaded loops, and at each iteration of one of them, a single data item is transferred to or from the ACM.

Classical semaphores can be used to protect write and read operations on a shared memory. However, if data items are large this approach does not provide a minimum locking between the writer and the reader [5]. This is because the time needed to perform the data access operations depends on the size of the data items. Single-bit *unidirectional* control variables allows the reduction of synchronization control to the reading and writing of them by extremely simple atomic actions [7]. Unidirectional variables are those that can only be modified by one of the processes. This provides a maximum asynchrony in particular, if the setting, resetting and referencing of control variables can be regarded as atomic events.

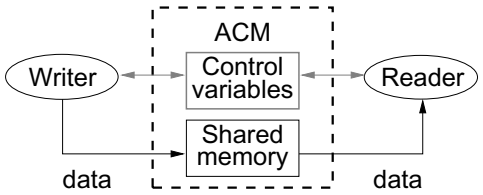

Figure 1. ACM with shared memory and control variables

ACMs are classified according to their *overwriting* and *re-reading* policies [7, 8]. Overwriting occurs when the ACM is full of data that has not been read before and the producer discards some of the existing data items in the buffer. Re-reading occurs when all data in the ACM has already been read by the consumer and it is allowed to re-read some item. In this way, any combination of those is allowed. Table 1 shows such a classification. `RRBB` denotes an ACM that only allows re-reading. On the other hand, the `OWBB` scheme allows only overwriting. Finally, the `OWRRBB` scheme allows both re-reading and overwriting while the `BB` scheme does not allow any of them.

	No re-reading	**Re-reading**
No overwriting	`BB`	`RRBB`
Overwriting	`OWBB`	`OWRRBB`

Table 1. Classification of ACMs

The choice of which class of ACM to use is based on data requirements and timing restrictions [5, 7]. For the re-reading class, it is more convenient to re-read the item

1522-4902/08 $25.00 © 2008 IEEE
DOI 10.1109/SCCC.2008.21

from the previous cycle. For overwriting, either the newest or the oldest item in the buffer can be overwritten [1, 7, 9]. Overwriting the newest item [9] attempts to provide the best continuity of data, which can also be achieved with a buffer of significant size. Overwriting the oldest item is based on the assumption that newer data is always more relevant than older.

One possible hazard in a binary unidirectional control variable is associated with metastability, which may happen when a control variable is modified and referenced about the same time by two asynchronous processes [4, 6]. A metastable binary variable may stay at an analogue value approximately midway between logic values 0 and 1 for an indefinite period of time, and it will eventually *"resolve"* to one of them non-deterministically. In practice, the effects of metastability can be minimized by adding a chain of flip-flops to the design reducing the probability of metastability. ACM algorithms operate correctly if their control variables are resolved before use.

1.1. An introductory example

Now consider an RRBB ACM with three data cells. The single-bit control variables r_i and w_i, with $i \in \{0, 1, 2\}$, are used to indicate which cell each process must access. Initially the reader is pointing at cell 0, $r_0 = 1$ and $r_1 = r_2 = 0$, and the writer is pointing at cell 1, $w_1 = 1$ and $w_0 = w_2 = 0$. The shared memory is initialized with some data. This configuration is shown in Figure 2.

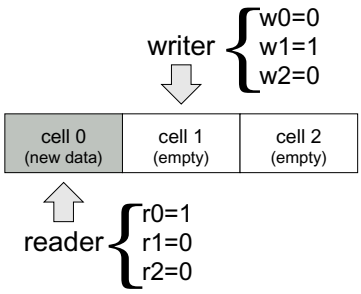

Figure 2. Execution of RRBB ACM with 3 cells

The writer first accesses the shared memory and then advances to the next cell, but only if the reader is not pointing at it. A possible trace for the writer is $\langle wr_1 wr_2 wr_0 wr_1 \rangle$, where wr_i denotes "*write data on cell i*". The reader first advances to the next cell if the writer is not there and then performs the data transfer, otherwise it re-reads the current cell. A possible trace for the reader is $\langle rd_0 rd_1 rd_1 rd_2 \rangle$.

In an RRBB ACM, no overwriting and allowing re-reading imply the following behavior:

- The writer first accesses the shared memory and then advances to the next cell, but only if the reader is not pointing at it;
- The reader first advances to the next cell if the writer is not there and then performs the data transfer, otherwise it re-reads the current cell.

1.2. ACMs properties

In general, and depending on how the read and write traces interleave, *coherence* and *freshness* properties must be satisfied.

Coherence is related to mutual exclusion between the writer and the reader. For example, a possible trace for this system is $\langle wr_1 wr_2 rd_0 \cdots \rangle$. After the writer executing twice, the next possible action for both processes is to access cell 0. This introduces the problem of data coherence when the reader and the writer are retrieving and storing data on the same memory locations.

Freshness is related to the fact that the last data record produced by the writer must be available for the reader. On the ACMs studied in this work, the reader always attempts to retrieve the oldest data stored in the shared memory that has not been read before. This means that the freshness property imposes a specific sequencing of data, i.e. the data is read in the same order that it is written. Depending on the ACM class, some data may be read more than once or may be missed. However, the sequence should be preserved. For the example above, one possible trace is $\langle wr_1 rd_0 wr_2 rd_1 rd_1 \cdots \rangle$. Note that at the moment the reader executes the first rd_1 action, the writer has already executed a wr_2. This means that there is some new data on cell 2. But the reader is engaged to execute rd_1 again, which violates freshness.

With a correct interleaving, both processes will avoid accessing the same data cell at the same time, the writer will not be allowed to overwrite unread data, and the reader will have the possibility of re-reading the most recent data only when there is no unread data in the ACM. For the example above, a correct trace is $\langle wr_1 rd_0 rd_1 wr_2 rd_1 wr_0 rd_2 wr_1 \rangle$. Observe that the sub-trace $rd_1 wr_2 rd_1$ does not contradict the fact that the reader only re-reads any data if there is no new one available. This is because after the first rd_1 there is no new data, then the reader prepares to re-read and from this point it will engage on a re-reading regardless the actions of the writer.

It is easy to note that any implementation that takes into account the use of binary control variables will be specific for ACMs of a certain size. For instance, in the example above 6 control variables are required. If the size increases to 4, another 2 variables are required. This means that more variables are needed when the size of the ACM grows,

which becomes to be more complex to be correctly implemented by a human. And for overwriting ACM classes it is necessary to have more control variables, and it is even more difficult to correctly deal with all of them.

In previous work a Petri net based method for the automatic synthesis of ACMs was presented [2]. That method receives as input a functional specification consisting of the ACM class its size. As output, it produces the C++ source code implementing the ACM as a shared memory with all the control needed.

In the current work that method is extended to support the generation of Verilog HDL source code. The specifics of the Verilog language are taken into account and it is shown how to obtain the ACM source code having only its functional specification as start point. On Section 2 the behavior expected from a hardware implementation of RRBBs ACMs is formally introduced. On Section 3, the basic design of those ACMs is introduced in the form of block diagrams and finite state machines, it is shown how RRBBs ACMs of any size can be synthesized. This design is used on Section 4 to outline a procedure for the automatic generation of Verilog code of RRBBs ACMs. Finally, on Section 5 the conclusions and future works are discussed.

2. Abstract model for RRBB ACMs

Previously the RRBB ACM has been formally specified as a transition system. However, that specification is adequate for the synthesis on software systems [2]. For this reason, in the current work, that specification has been extended to support hardware synthesis. An ACM state is defined by the data items available for reading. For each state, σ defines the queue of data stored in the ACM. More specifically, σ is a sequence $a_0 a_1 \cdots a_{j-1} a_j$, with $j < n$, where n is size of the ACM. The data item a_j is the last written data, and a_0 is the next data to be retrieved by the reader. The size of the ACM is given by its number of cells, i.e. the maximum number of data items the ACM can store at a certain time.

The data queue σ must also express if the processes are accessing the ACM or not. This is done by adding flags to the a_0 and a_j items. a_j^w indicates that the writer is starting to store a new data item. $a_j^{w'}$ indicates that it has finished writting the new data but has not released the cell yet. In both cases the data is not available for reading. Finally, a_j indicates that the item is available for reading and that the writer is ready to receive a new request. Similarly, a_0^r indicates that the reader has started consuming data a_0, $a_0^{r'}$ indicates that the reader has finished but has not released the cell yet, and a_0 indicates that the reader is ready to receive a new request.

Observe that σ can be interpreted as a stream of data that is passed from the writer (on the left) to the reader (on the right). There are six events that change the state of the ACM:

1. rd_r: reader receives a request to read a new data;
2. rd_b: reading a data item begins;
3. rd_e: reading a data item ends;
4. wr_r: writer receives a request to write a new data;
5. wr_b: writing a data item begin;
6. wr_e: writing a data item ends.

The notation $\langle \sigma_i \rangle \xrightarrow{e} \langle \sigma_j \rangle$ denotes the occurrence of event e from state $\langle \sigma_i \rangle$ to state $\langle \sigma_j \rangle$, whereas $\langle \sigma \rangle \xrightarrow{e} \perp$ is used to denote that e is not enabled in $\langle \sigma \rangle$.

In RRBB ACMs, the reader is required not to wait when starting an access to the ACM. In the case there is no new data in the ACM, the reader will re-read some data that was read before.

The writer can add data in the ACM until it is full. In such case, the writer is required to wait until the reader retrieves some data from the ACM. The reader always tries to retrieve the oldest non-read data and, if all data in the ACM has been read before, then it attempts to re-read the last retrieved data item.

Definition 1 formally captures the behavior of RRBB ACMs. Rules 1-4 model the behavior of the writer. Rules 5-9 model the behavior of the reader.

Definition 1 (RRBB transition rules) *The behavior of an RRBB ACM is defined by the following set of transitions (n is the number of cells of the ACM and the cells are numbered from 0 to $n-1$):*

1. $\langle \sigma \rangle \xrightarrow{wr_r} \langle \sigma a^w \rangle$
2. $\langle \sigma a^w \rangle \xrightarrow{wr_b} \langle \sigma a^{w'} \rangle$
3. $\langle \sigma a^{w'} \rangle \xrightarrow{wr_e} \langle \sigma a \rangle \quad \textbf{if} \ \ |\sigma| < n$
4. $\langle \sigma a^{w'} \rangle \xrightarrow{wr_b(a)} \perp \quad \textbf{if} \ \ |\sigma| = n$
5. $\langle a \sigma \rangle \xrightarrow{rd_r} \langle a^r \sigma \rangle$
6. $\langle a^r \sigma \rangle \xrightarrow{rd_b} \langle a^{r'} \sigma \rangle$
7. $\langle a^{r'} \sigma \rangle \xrightarrow{rd_e} \langle \sigma \rangle \quad \textbf{if} \ \ |\sigma| > 0 \ \wedge \ \sigma \neq b^w$
8. $\langle a^{r'} \rangle \xrightarrow{rd_e} \langle a \rangle$
9. $\langle a^{r'} b^w \rangle \xrightarrow{rd_e} \langle a b^w \rangle$

Rule 1 models the start of a write action for a new data item a and signaling that it is not available for reading (a^w). Rule 2 models the completion of the write action, however the item is not released for reading yet. Rule 3 models the act of making the new data available for reading. Finally, rule 4 represents the blocking of the writer when the ACM is full ($|\sigma| = n$).

Rule 5 models the beginning of a read action retrieving data item a and indicating that it is being read (a^r). Rule 6 models the completion of the read operation but the accessed cell is not released yet. Rules 7 to 9 model the reader releasing the cell and preparing to receive a new request. On Rule 7, a is removed from the buffer when other data is available. On the other hand, rules 8 and 9 model the action of releasing a cell release when no more data is available for reading. In this case, the data is not removed from the buffer and is available for re-reading. This is necessary due to the fact that the reader is required not to be blocked even if there is no new data in the ACM.

It is important to observe that in the state $\langle a^r b^w \rangle$ the next element to be retrieved by the reader will depend on the order that events $wr_e(b)$ and $rd_e(a)$ occur. If the writer delivers b before the reader finishes retrieving a, then b will be the next data to be read. Otherwise, the reader will prepare to re-read a.

3. Design for RRBB ACMs

In this Section the design of a 3-cells RRBB ACM will be discussed. This design is for an specific number of cells. However it is not difficult to extend it to support an arbitrary ACM size. The general structure for an ACM hardware implementation is introduced by the block diagram on Figure 3. The ACM is composed by three modules: i)writer module; ii)reader module and iii) shared memory module. On Figure 3 the basic architecture of the ACM is shown. Besides `clock` and `reset` signals, there are a number of input and output signals whose purpose is to provide communication between the writer and reader processes. Specifically, the writer process can send a request signal (`w_req`) to the ACM, the new data (`w_data[]`) to be written and receive an acknowledgment signal (`w_ack`) as response to this request. On the other hand, the reader process can send a request signal (`r_req`) and receive as response a new data item (`r_data[]`) and an acknowledgment signal (`r_ack`).

Each module of the ACM communicates with the others using proper internal signals. For instance, when the writer module receives a request, indicated by `w_req=1`, it first makes a request to the shared memory module setting `req` to 1 and then it waits for the `ack` signal. After receiving the `ack`, it forwards the signal to the writer process using the wire `w_ack` and then it checks if the reader module is accessing the next cell by setting `sel_rd[]` to the value of the next cell and checking the value of the input signal `res_rd`. In the negative case, the new data is released for reading and the writer module prepares to read the next cell. Otherwise it waits until the reader is not pointing to the next cell any more. The behavior of the reader is similar to the behavior of the writer, except by the fact the if the writer module is pointing to the next cell, the reader prepares to re-read the current cell.

The writer module is detailed in the block diagram of Figure 4. The control variables of the writer are implemented by the flip-flops named `wr0`, `wr1` and `wr2` on the left side of the diagram. Each flip-flop implements one control variables and they are one-hot encoded, meaning that one of them, and only one, must have value set to 1 at any time. At each clock tick, the writer engine updates the values of the control variables if necessary.

The reader module, which is shown in the block diagram of Figure 5, has access to the control variables of the writer through a multiplexor. It is basically equal to the writer module. The main difference is that it receives data from the shared memory module and returns this data to the reader process. Observe that in both cases, a module *"asks"* the other if a cell is being accessed or not by setting the value of the select signal properly. The answer comes by the corresponding result signal. In any case, the result is only perceived after passing through two sequential flip-flops clock signals, which requires two clock signals. This is necessary in order to minimize the probability of metastability problems since the access to the control variables is not controlled by any mutual exclusion mechanism.

Finally, the engine of each module are defined as Finite State Machines (FSM). In Figure 6 the behavior of the writer is modeled. Initially the writer is on idle state ready to access the cell number 1, state labeled **idle1**. It is already pointing to cell 1 and the next cell has been selected (`sel=2`). When a writing request is received (`ereq=1`) the state changes to **init1**, and a request is made to the shared memory module (`req=1`). When the request is executed (`ack=1`) the state changes to **end1**, the received data is returned to the calling process and the engine checks if the reader is accessing the next cell by testing `!rd`. If the reader is not accessing that cell, then the writer advances to it updating its own control variables, sends an acknowledgment signal to the served process and checks one cell ahead for the reader. This is indicated by `addr=2`, `w1=0`, `w2=1`, `eack=1` and `sel=0`. This cycle repeats until the writer returns to the state **idle1**.

The reader module works on the same way as the writer module. The FSM of the reader engine is shown in Figure 7. The main difference compared to the writer FSM on Figure 6 is that in order to finish a data access action, while the writer blocks if the reader is pointing to the next cell,

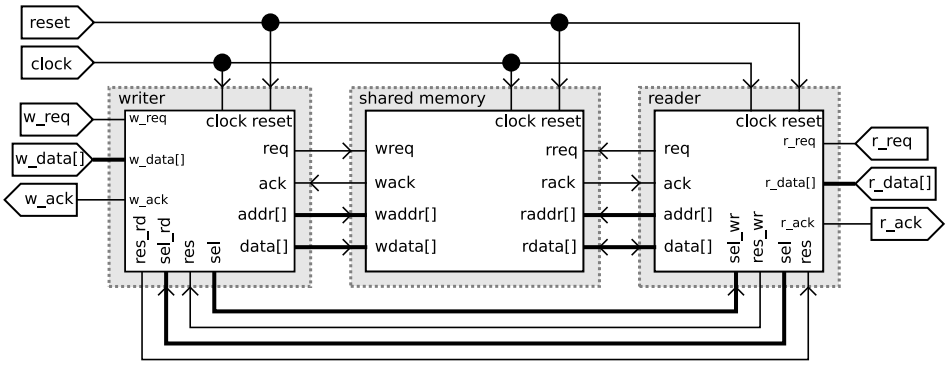

Figure 3. The ACM block diagram

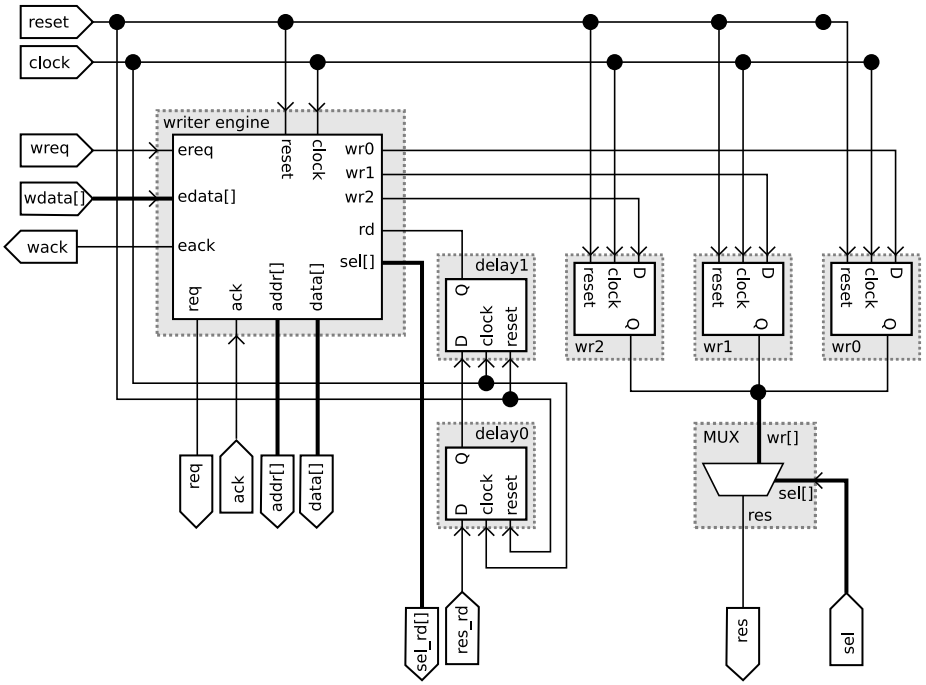

Figure 4. The writer block diagram

the reader does not block. Instead, it returns to the state in which it is ready to access the same cell it has just read. In other words, it prepares to re-read the current cell and the values of the control variables are not modified. Note the arcs with conditions `!wr` and `wr` starting from any state label **endN** with N={0,1,2}. These arcs indicate that the status of the writer does not block the reader. The mechanism to access the control variable of the writer module is the same and it communicates with the shared memory module in the same way as the writer.

Finally, on Figure 8 it is shown the FSM for the shared memory module. This module is actually responsible for executing read and write operations in the communication buffer, while the reader and writer modules control where these operations are done. The shared memory module communicates with both writer and reader modules. It receives a request from the writer, a data to be stored and the address to store the data. After saving the data it returns an acknowledgment indicating termination of the action. It also receives a request and an address from the reader, and returns the data requested and an acknowledgment signal. Observe that writer and read operations can be handled concurrently.

From Figures 6 and 7 it is easy to observe that each FSM

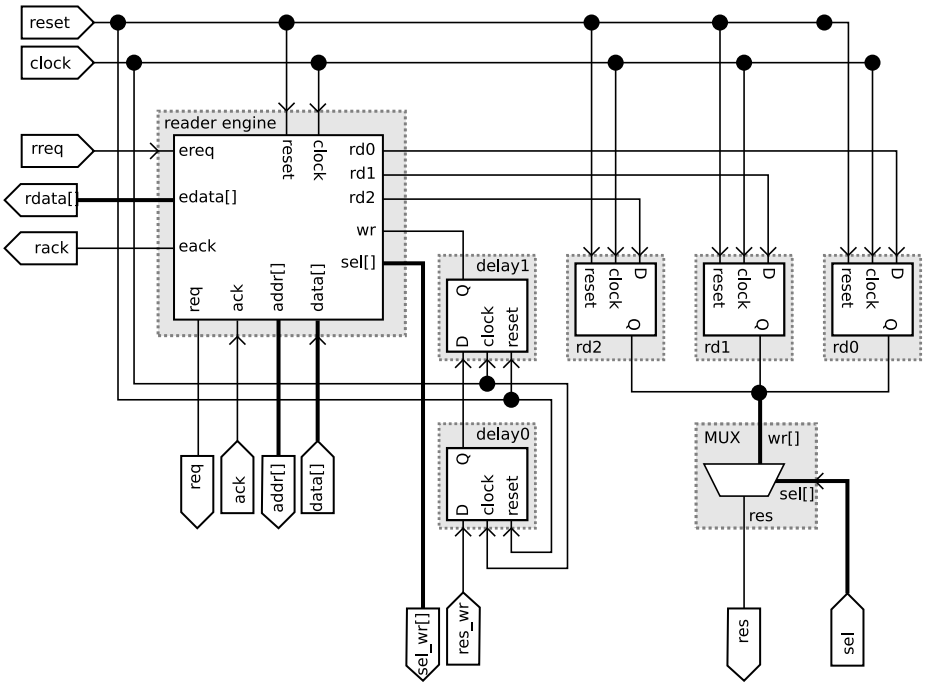

Figure 5. The reader block diagram

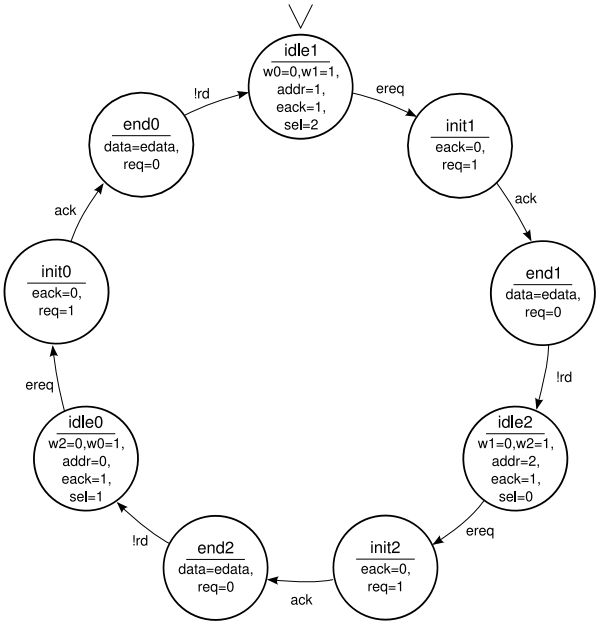

Figure 6. The writer finite state machine

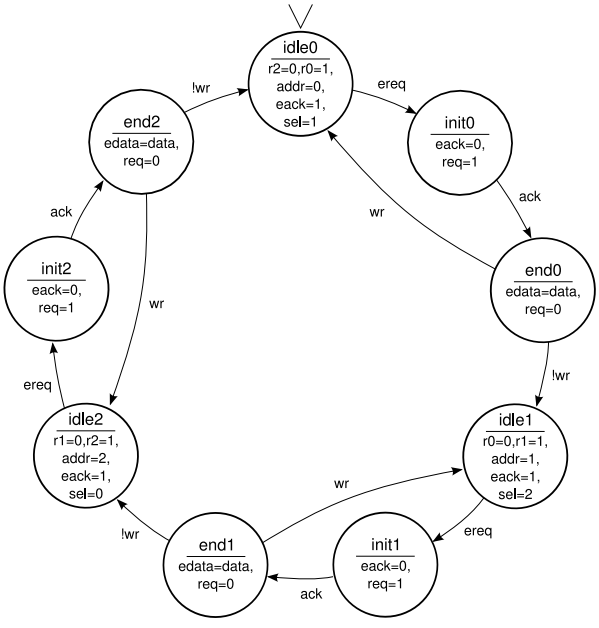

Figure 7. The reader finite state machine

presents a basic structure that is repeated into the entire model. This happens due to the fact that the control over each ACM cell is exactly the same, differing only on the addresses they control the access to. For instance, we can be observe that the writer FSM can be easily obtained from the FSM module shown in Figure 9.

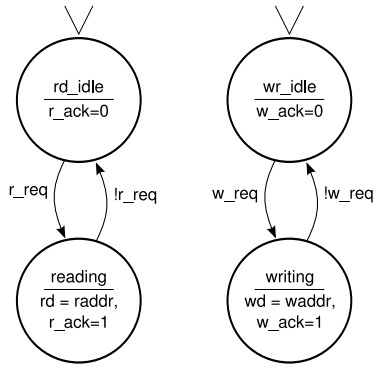

Figure 8. The shared memory finite state machine

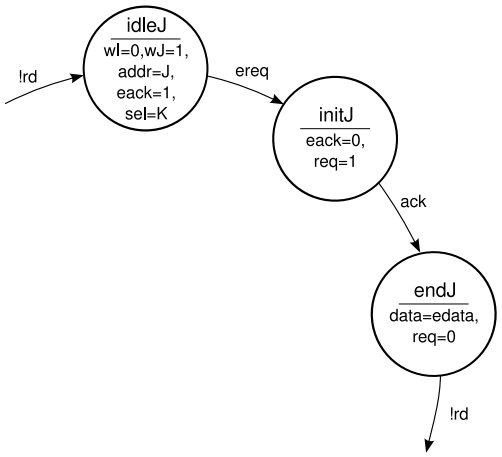

Figure 9. FSM writer module

More specifically, in order to obtain the FSM of the 3-cell RRBB ACM described previously, it is only needed to instantiate a number n of modules like the one on Figure 9 and connect them properly. Since this module express all the control needed for one ACM cell, the total amount of modules needed corresponds to the size of the ACM. So, for the 3-cell RRBB above, three modules are needed. To instantiate a module for the j^{th} cell it is necessary to generate an FSM of the module and replace all occurrences of **I**, **J** and **K** properly. Note that **I**, **J** and **K** represent the number of the previous, the current and the next cell respectively. And they must be replaced by $j-1$, j and $j+1$ respectively.

Finally it is necessary to connected the modules obtained. This is easily done by just connecting the output arc labeled `!rd` of the j^{th} module to the input arc labeled `!rd` of the $(j+i)^{th}$ module. Obverse that the we are considering the operation $(j+1)$ as $((j+1) \mod n)$, where n is the ACM size. After these two simple steps the FSM on the RRBB ACM is generated.

The same procedure can be used to obtain the FSM for the reader process. The FSM module for the reader is shown in Figure 10.

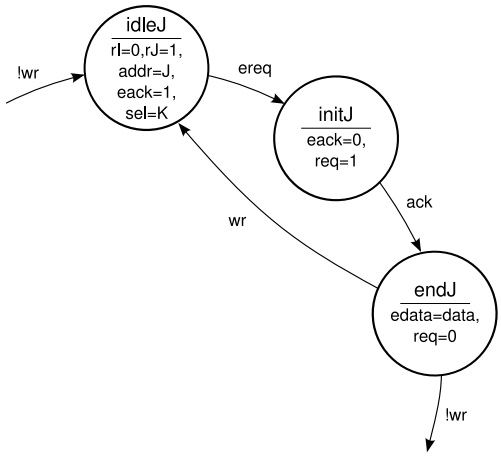

Figure 10. FSM reader module

4. Verilog code synthesis

To complete the hardware synthesis for RRBB ACMs introduced on this work, it is necessary to point out how to obtain an artifact, that can be synthesized into a physical hardware, from the set of FSMs presented on Section 3. On this Section it will be outlined how to obtain a Verilog code for RRBB ACMs.

The block diagrams described on Section 3 ca be used to generate a set of templates for RRBBs ACMs of any size. It is only necessary to take care to setup correctly the datatype to be transmitted and the size of the ACM. Note that in the ACM module this size does not appear explicitly, however the signal `sel[]` depends on it. More specifically, this signal should have $log_2 n - 1$ wires, if n is the size of the ACM. On the reader and the writer modules, besides that, it is also necessary to instantiate a number of flip-flops corresponding to the size of the ACM. Each flip-flop will corresponds to a binary variable that controls the access to an specific cell. Also, all wires should be correctly connected. Since this is almost static, it will not be addressed here.

The main problem is on the synthesis of the code that controls the access to to the shared memory. This is encapsulated in the writer engine and reader engine modules. These are described by the FSMs introduced in Section 3. The Verilog code generation uses the simple idea of getting each FSM module used to build the writer engine or the reader engine and generate a piece of Verilog code from

a template that is equivalent to the FSM module. For the writer, its FSM module is mapped into the into the following piece of Verilog code:

```
1  state[IDLE_J]: begin
2    if (ereq) begin
3      eack <= 1'b0;
4      req <= 1'b1;
5      state[IDLE_J] <= 1'b0;
6      state[INIT_J] <= 1'b1;
7    end
8  end
9  state[INIT_J]: begin
10   if (ack) begin
11     req <= 1'b0;
12     data <= edata;
13     state[INIT_J] <= 1'b0;
14     state[END_J] <= 1'b1;
15   end
16 end
17 state[END_J]: begin
18   if (!rd) begin
19     cell_J <= 1'b0;
20     cell_J+1 <= 1'b1;
21     sel <= J+2;
22     eack <= 1'b1;
23     addr <= 1;
24     state[END_J] <= 1'b0;
25     state[IDLE_J+1] <= 1'b1;
26   end else begin
27     state[END_J] <= 1'b1;
28   end
29 end
```

So, for each FSM module instantiated, the corresponding Verilog code template above should also be instantiated. To proceed with this step, it is necessary to take care of replacing the Js properly. In the above, the J should be replaced by the number j of the j^{th} cell. Note that J+1 should be replaced by $((j+1) \mod n)$ and J+2 by $((j+2) \mod n)$, where n is the size of the ACM. The source code obtained implies two things:

1. All states of the FSM are enumerate and there is an state array that is one-hot encoded to indicate the current state;

2. There is a **case** statement in which the code above is inserted.

For instance, for the 3-cell RRBB ACM, the state enumeration is done by:

```
1  parameter IDLE_0=0, INIT_0=1, END_0=2,
2            IDLE_1=3, INIT_1=4, END_1=5,
3            IDLE_2=6, INIT_2=7, END_2=8;
```

and the **case** statement is defined by:

```
1  case (1'b1)
2    CASE BODY GENERATED FROM FSM
3  endcase
```

Observe that line 2 should be replaced by the proper case statements. The source code obtained for the reader engine is similar to the one obtained to the writer, and its generation follows the same idea. Many details related to the target programming language have been omitted here, but they are not crucial to the comprehension of the approach.

The procedure described above has been implemented [1] and used to generate a number of RRBBs ACMs of different size. The code synthesized has been submitted to 1.000.000.000 simulation cycles and, in all cases, no violations of freshness and coherence properties were encountered.

5. Conclusions and future work

On this work the hardware synthesis problem for rereading asynchronous communications mechanisms has been addressed. The method presented here is based on the use of modules to the generation of FSMs specifications for each ACM process.

Firstly, the behavior of RRBB ACMs was formally defined and the basic properties they must satisfy were discussed. Then the basic design of an RRBB ACM was described. The main block diagrams were introduced and the FSMs specifying the control engine of each process were defined. Then it has been shown how to obtain a procedure that can be used to generate the FSM engines for RRBBs ACMs of any size. And, finally, it has been introduced how to translate the block diagrams and FSMs into Verilog code that can be used to generate hardware or for simulation purposes.

The method above has been implemented and made available for public download. A different number of ACMs have been generated and in all cases coherence and freshness has been analyzed through simulation. However, the method lacks a formal proof of its correctness, even there are strong evidences of it.

This work extends previous ones [3, 2] by adding support for hardware synthesis. This is done in two ways. First, the formal behavior of RRBBs ACMs is extend to consider hardware implementations and not only software implementations. Second, the automatic procedure for generating Verilog code of RRBBs ACMs is defined and implemented.

Next steps to complete automation of ACMs generation includes the support for the generation of the overwriting ACM policies. And the presentation of a formal proof of

[1] See ACMgen tool at `http://sourceforge.net/project/acmgen/`.

its correctness. Also, a number of interesting applications demonstrating its usefulness need to be introduced.

Acknowledgments

The authors benefited from extensive discussions with Alex Yakovlev and Fei Xia and wish to express out their gratitude.

References

[1] J.-P. Fassino. *THINK: vers une architecture de systèmes flexibles*. PhD thesis, École Nationale Supérieure des Télécommunications, Dec. 2001.

[2] K. Gorgônio, J. Cortadella, and F. Xia. A compositional method for the synthesis of asynchronous communication mechanisms. In J. K. an Alex Yakovlev, editor, *ICATPN*, number 4546 in LNCS, pages 144–163. Springer-Verlag Berlin Heidelberg, 2007.

[3] K. Gorgônio, J. Cortadella, F. Xia, and A. Yakovlev. Automating synthesis of asynchronous communication mechanisms. *Fundamenta Informaticae*, 78(1):75–100, June 2007.

[4] L. Kleeman and A. Cantoni. On the unavoidability of metastable behavior in digital systems. *IEEE Transactions on Computers*, 36(1):109–112, 1987.

[5] L. Lamport. On interprocess communication — parts I and II. *Distributed Computing*, 1(2):77–101, 1986.

[6] L. R. Marino. General theory of metastable operation. *IEEE Transactions on Computers*, 30(2):107–115, 1981.

[7] H. R. Simpson. Protocols for process interaction. *IEE Proceedings on Computers and Digital Techniques*, 150(3):157–182, May 2003.

[8] F. Xia, F. Hao, I. Clark, A. Yakovlev, and G. Chester. Buffered asynchronous communication mechanisms. In *Proceedings of the Fourth International Conference on Application of Concurrency to System Design (ACSD'04)*, pages 36–44. IEEE Computer Society, 2004.

[9] A. Yakovlev, D. J. Kinniment, F. Xia, and A. M. Koelmans. A fifo buffer with non-blocking interface. *TCVLSI Technical Bulletin*, pages 11–14, Fall 1998.